CS 378 - Autonomous Vehicles in Traffic I
Week 6b - Velodyne Processing, rosbag, rviz

Today

- Today we'll see how to use Velodyne data
- We'll take a look at our obstacle detection code
- We'll look at recording data using the rosbag tool and playing it back
- We'll take a look at the main visualization software in ROS - rviz.

Announcements

- Programming assignment 2 due tonight
- I will try and discuss some of the common problems faced by students in the first 2 programming assignments sometime next week
- Programming Assignment 3 should be up by tonight. I'll discuss it next week on Monday
- Make sure you have commit permissions and svn commit is working at least a few hours before the deadline.
- Possible car visit on Tuesday (2/28). I will coordinate with you guys on Monday. We will also have the car on campus (Parking lot north of ACE building) for Explore UT on Saturday (3/3). I am looking for 2-3 volunteers, will send email sometime tonight.

Code reorganization

- Jack has been reorganizing code in the Velodyne stack to create a cleaner interface. Some of these changes are still in progress.
- To get an updated copy of the code, run the svn update command.
  ○ cd ~/svn/utexas-art-ros-pkg
  ○ svn up
  ○ rosmake art_run
- You will need to do this to get the examples from today
The velodyne_driver package

- The driver gets raw packet data from the Velodyne, and transmits it in raw form inside the ROS ecosystem.
- This data is still needs to be processed to get a pointcloud like the one you saw on Monday.
- We record this raw data, as it uses up far less space than a pointcloud (more about recording data in later slides).


The velodyne_pointcloud package

- It accepts the raw data and publishes out the information in a well defined point cloud structure that can be understood by ROS nodes.
- The cloud_node node subscribes to:
  - /velodyne_packets [velodyne_msgs/VelodyneScan]
- It publishes:
  - /velodyne_points [sensor_msgs/PointCloud2]
- This is the same message type that was being published by some node during Monday's demo.
- Aside: Why is it called PointCloud2?
  - Nothing is perfect. The earlier format [sensor_msgs/PointCloud] was quite inefficient.
The **cloud_node** node

- Running this node:
  - `rosrun velodyne_pointcloud cloud_node`

- You will have to run this node when playing back recorded data
  - We store `/velodyne_packets`, not `/velodyne_points`

- OK, so we have this node publishing point clouds. How do I use it?
  - Let's take a look at the example code provided for this
  - You will to update your code to get this example:
    - `roscd art_examples`
    - `svn up`

---

Writing a subscriber

- I have a very basic subscriber in the `velodyne_example` package
  - `roscd velodyne_example`
  - `gedit src/subscriber.cc`

- Running the subscriber
  - `rosrun velodyne_example subscriber`

---

### subscriber.cc

```cpp
#include <ros/ros.h>
#include <pcl_ros/point_cloud.h>
#include <pcl/point_types.h>
#include <boost/foreach.hpp>

typedef pcl::PointXYI VPoint;
typedef pcl::PointCloud<VPoint> VPointCloud;

void callback(const VPointCloud::ConstPtr &msg) {
    ROS_INFO("Cloud: width = %d, height = %d", msg->width, msg->height);
    /** A good way to access all the points */
    //BOOST_FOREACH(const VPoint& point, msg->points) {
    /** Access point values using */
    // point.x
    // point.y
    // point.z
    // point.intensity
    //}
}

int main(int argc, char** argv) {
    ros::init(argc, argv, "velodyne_example_sub");
    ros::NodeHandle nh;
    ros::Subscriber sub =
    nh.subscribe<VPointCloud>("/velodyne_points", 1, callback);
    ros::spin();
    return 0;
}
```

---

Hold on!

- There was a bit of trickery in the last few slides
  - The code in the last slide was subscribing to
    `pcl::PointCloud<pcl::PointXYZI>`
  - In earlier slides I said I was publishing a message type called `sensor_msgs::PointCloud2`
  - ?!??!?

- The truth is that using `sensor_msgs::PointCloud2` is an efficient data structure (good for transport) but not so user friendly. It is also ROS aware.
- `pcl::PointCloud` is an easy to use data structure that is used by the PCL. It is independent of ROS.
- So how were we able to subscribe to the PCL datatype?
Enter `pcl_ros`

![Diagram](image)

**Using the `pcl_ros` bridge**

- To use the bridge, include the `pcl_ros` header
  - `#include <pcl_ros/point_cloud.h>`
- If you are working independent of ROS, the following header is sufficient
  - `#include <pcl/point_cloud.h>`
- OK, so this was using `pcl_ros` while subscribing. Does it work for publishing as well?
  - Required for programming assignment 3

**Height-Difference Map**

- Height-difference maps identify vertical surfaces in the environment.
  - Without the need for computationally intensive algorithms for 3D, real-time modeling
- At each cycle (i.e. every complete set of 360° data) we create a 2D (x, y) grid map from the 3D point cloud.
  - Record the maximum and minimum z (vertical) values seen in each grid cell.
- Next a simulated lidar scan is produced from the 2D grid.
  - Cast rays from the sensor origin, an obstacle is detected whenever the difference between the max and min z values is above a threshold
- The result is a 360° 2D simulated lidar scan, which looks like and can be further processed in a manner similar to the data outputted by SICK lidar devices

**Algorithm 1 Height-Difference Map**

```plaintext
1: for all p ∈ pointcloud do
2:   𝑥, 𝑦 ← grid.index(p.x, p.y)
3:   grid[x,y].z_min ← min(p.z, grid[x][y].z_min)
4:   grid[x,y].z_max ← max(p.z, grid[x][y].z_max)
5: end for
6: for all angle ∈ (0, 360) do
7:   lidar.scan[angle] = MAX_DISTANCE
8:   for all cell ∈ raytrace(angle) do
9:     if (cell.z_max - cell.z_min) > THRESHOLD then
10:        lidar.scan[angle] = cell.distance
11:        break
12:     end if
13: end for
14: end for
```
The *robag* tool

- As some of you may have realized by now, writing a lot of code at the car is not feasible.
- Lucky for us, we have a way of collecting data easily.
- You can record topics using the robag record command.
- Examples:
  - `robag help record`
  - `robag record /velodyne_packets`
  - `robag record /velodyne_packets /odom`
  - `robag record -a`
- Recording a bag will output a *bag* file

Playing back recorded data

- robag provides a playback tool
- Examples:
  - `robag play <bag_file>`
  - `robag play <bag_file> /velodyne_packets`
  - `robag play *_05.bag`
  - `robag play *_05.bag -l --clock` (Recommended)
- This is *almost* as good as having all these messages being produced live
Putting it all together

- To run the entire chain, follow these steps:
  - Get a bag file
  - Start roscore
  - Play the bag file
  - Start `cloud_node` (for bags before 2/22)
    - `roslaunch velodyne_pointcloud cloud_node velodyne_packets:=velodyne/packets`
  - (For bags after 2/22)
    - `roslaunch velodyne_pointcloud cloud_node`
  - Visualizing in rviz
    - `roslaunch rviz rviz`
  - Build the subscriber
    - `rosmake velodyne_example`
  - Running the subscriber
    - `roslaunch velodyne_example subscriber`

rviz - The ROS visualization tool

- We briefly saw `rviz` on Monday.
- An excellent tutorial for navigating `rviz` is available here
- I will run a live demo of rviz in class today.
- There are a couple of things (like `tf`) that we’ll explain later in the semester.
  - For the time being, set the following values when playing back a bag file:
    - Fixed frame: `/velodyne`
    - Target frame: `<Fixed frame>`